

SYSTEM AND METHOD FOR EXECUTING WIRELESS COMMUNICATIONS DEVICE DYNAMIC INSTRUCTION SETS

BACKGROUND OF THE INVENTION

5 1. Field of the Invention

This invention generally relates to wireless communications devices and, more particularly, to a system and method for executing dynamic instructions sets with the system software of a wireless communication device in the field.

10 2. Description of the Related Art

It is not uncommon to release software updates for phones that are already in the field. These updates may relate to problems found in the software once the phones have been manufactured and distributed to the public. Some updates may
15 involve the use of new features on the phone, or services provided by the service provider. Yet other updates may involve regional problems, or problems associated with certain carriers. For example, in certain regions the network layout of carriers may impose airlink interface conditions on the handset that cause the handset to
20 demonstrate unexpected behavior such as improper channel searching, improper call termination, improper audio, or the like.

The traditional approach to such updates has been to recall the wireless communications device, also referred to herein as a wireless device, phone, telephone, or handset, to the nearest carrier
25 retail/service outlet, or to the manufacturer to process the changes. The costs involved in such updates are extensive and eat into the bottom line. Further, the customer is inconvenienced and likely to be

irritated. Often times, the practical solution is to issue the customer new phones.

The wireless devices are used in a number of environments, with different subscriber services, for a number of different customer applications. Therefore, even if the software of a wireless device can be upgraded to improve service, it is unlikely that the upgrade will provide a uniform improvement for all users.

It would be advantageous if wireless communications
device software could be upgraded cheaply, and without
10 inconvenience to the customer.

It would be advantageous if wireless communications device software could be upgraded without the customer losing the use of their phones for a significant period of time.

It would be advantageous if wireless communications
15 device software could be updated with a minimum of technician
service time, or without the need to send the device into a service
facility.

It would be advantageous if the wireless device system software could be differentiated into code sections, so that only specific code sections of system software would need to be replaced, to update the system software. It would be advantageous if these code sections could be communicated to the wireless device via the airlink.

It would be advantageous if the code section updates
could be made uniquely for each wireless communications device
25 based upon that device's circumstances.

It would be advantageous if the wireless device could monitor the performance of the wireless device system software, collect performance data, and transmits the data to a system central collection depot for analysis.

5

SUMMARY OF THE INVENTION

Wireless communications device software updates give customers the best possible product and user experience. An expensive component of the business involves the recall of handsets to update the software. These updates may be necessary to offer the user additional services or to address problems discovered in the use of the phone after it has been manufactured. The present invention makes it possible to practically upgrade handset software in the field, via the airlink interface. More specifically, the present invention permits the wireless communication device to execute dynamic instruction sets. These dynamic instruction sets permit the wireless device to "intelligently", or conditionally upgrade the system software and system data. Further, the dynamic instruction sets permit the wireless device to monitor system software performance, and transmit the performance data for analysis.

Accordingly, a method is provided for executing dynamic instruction sets in a wireless communications device. The method comprises: forming the system software into symbol libraries, each symbol library comprising symbols having related functionality; arranging the symbol libraries into code sections in a code storage section nonvolatile memory; executing system software; receiving a

5

10

BRIEF DESCRIPTION OF THE DRAWING

device software maintenance system.

15

20

memory.

table of Fig. 3.

25

3, with symbols.

Fig. 7 is a table representing the symbol offset address table of Fig. 3.

Fig. 8 is a depiction of the operation code (op-code) being accessed by the run-time engine.

5 Fig. 9 is a more detailed depiction of the first operation code of Fig. 8.

Figs. 10a and 10b are flowcharts illustrating the present invention method for executing dynamic instruction sets in a wireless communications device.

10 Fig. 11 is a flowchart illustrating an exemplary dynamic instruction set operation.

Fig. 12 is a flowchart illustrating another exemplary dynamic instruction set operation.

15 Fig. 13 is a flowchart illustrating a third exemplary dynamic instruction set operation.

Fig. 14 is a flowchart illustrating a fourth exemplary dynamic instruction set operation.

Fig. 15 is a flowchart illustrating a fifth exemplary dynamic instruction set operation.

20

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Some portions of the detailed descriptions that follow are presented in terms of procedures, steps, logic blocks, codes, processing, and other symbolic representations of operations on data bits within a wireless device microprocessor or memory. These
25 descriptions and representations are the means used by those skilled

5 instructions leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a microprocessor based wireless device. It
10 has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. Where physical devices, such as a memory are mentioned, they are connected to other physical devices through a bus or other electrical connection. These physical
15 devices can be considered to interact with logical processes or applications and, therefore, are “connected” to logical operations. For example, a memory can store or access code to further a logical operation, or an application can call a code section from memory for execution.

20 It should be borne in mind, however, that all of these and
similar terms are to be associated with the appropriate physical
quantities and are merely convenient labels applied to these
quantities. Unless specifically stated otherwise as apparent from the
following discussions, it is appreciated that throughout the present
25 invention, discussions utilizing terms such as "processing" or
"connecting" or "translating" or "displaying" or "prompting" or

"determining" or "displaying" or "recognizing" or the like, refer to the action and processes of in a wireless device microprocessor system that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and
5 memories into other data similarly represented as physical quantities within the wireless device memories or registers or other such information storage, transmission or display devices.

Fig. 1 is a schematic block diagram of the overall wireless device software maintenance system 100. The present invention
10 system software organization is presented in detail below, following a general overview of the software maintenance system 100. The general system 100 describes a process of delivering system software updates and instruction sets (programs), and installing the delivered software in a wireless device. System software updates and patch
15 manager run time instructions (PMRTI), that are more generally known as instruction sets or dynamic instruction sets, are created by the manufacturer of the handsets. The system software is organized into symbol libraries. The symbol libraries are arranged into code sections. When symbol libraries are to be updated, the software
20 update 102 is transported as one or more code sections. The software update is broadcast to wireless devices in the field, of which wireless communications device 104 is representative, or transmitted in separate communications from a base station 106 using well known, conventional air, data or message transport protocols. The invention
25 is not limited to any particular transportation format, as the wireless communications device can be easily modified to process any

available over-the-air transport protocol for the purpose of receiving system software and PMRTI updates.

The system software can be viewed as a collection of different subsystems. Code objects can be tightly coupled into one of these abstract subsystems and the resulting collection can be labeled as a symbol library. This provides a logical breakdown of the code base and software patches and fixes can be associated with one of these symbol libraries. In most cases, a single update is associated with one, or at most two, symbol libraries. The rest of the code base, the other symbol libraries, remains unchanged.

The notion of symbol libraries provides a mechanism to deal with code and constants. The read-write (RW) data, on the other hand, fits into a unique individual RW library that contains RAM based data for all libraries.

Once received by the wireless device 104, the transported code section must be processed. This wireless device over-writes a specific code section of nonvolatile memory 108. The nonvolatile memory 108 includes a file system section (FSS) 110 and a code storage section 112. The code section is typically compressed before transport in order to minimize occupancy in the FSS 110. Often the updated code section will be accompanied by its RW data, which is another kind of symbol library that contains all the RW data for each symbol library. Although loaded in random access volatile read-write memory 114 when the system software is executing, the RW data always needs to be stored in the nonvolatile memory 108, so it can be loaded into random access volatile read-write memory 114 each time

the wireless device is reset. This includes the first time RW data is loaded into random access volatile read-write memory. As explained in more detail below, the RW data is typically arranged with a patch manager code section.

- 5 The system 100 includes the concept of virtual tables. Using such tables, symbol libraries in one code section can be patched (replaced), without breaking (replacing) other parts of the system software (other code sections). Virtual tables execute from random access volatile read-write memory 114 for efficiency purposes.
- 10 A code section address table and symbol offset address table are virtual tables.

- The updated code sections are received by the wireless device 104 and stored in the FSS 110. A wireless device user interface (UI) will typically notify the user that new software is
- 15 available. In response to UI prompts the user acknowledges the notification and signals the patching or updating operation. Alternately, the updating operation is performed automatically. The wireless device may be unable to perform standard communication tasks as the updating process is performed. The patch manager code
- 20 section includes a non-volatile read-write driver symbol library that is also loaded into random access volatile read-write memory 114. The non-volatile read-write driver symbol library causes code sections to be overwritten with updated code sections. The patch manager code section includes the read-write data, code section address table, and
- 25 symbol offset address table, as well a symbol accessor code and the symbol accessor code address (discussed below). Portions of this data

09927331-031001

are invalid when updated code sections are introduced, and an updated patch manager code sections includes read-write data, a code section address table, and a symbol offset address table valid for the updated code sections. Once the updated code sections are loaded
5 into the code storage section 112, the wireless device is reset.

Following the reset operation, the wireless device can execute the updated system software. It should also be understood that the patch manager code section may include other symbol libraries that have not been discussed above. These other symbol libraries need not be
10 loaded into read-write volatile memory 114.

Fig. 2 is a schematic block diagram of the software maintenance system 100, highlighting the installation of instruction sets via the airlink interface. In addition to updating system software code sections, the maintenance system 100 can download and install
15 dynamic instructions sets, programs, or patch manager instruction sets (PMIS), referred to herein as patch manager run time instructions (PMRTI). The PMRTI code section 200 is transported to the wireless device 104 in the same manner as the above-described system software code sections. PMRTI code sections are initially stored in the
20 FSS 110. A PMRTI code section is typically a binary file that may be visualized as compiled instructions to the handset. A PMRTI code section is comprehensive enough to provide for the performance of basic mathematical operations and the performance of conditionally executed operations. For example, an RF calibration PMRTI could
25 perform the following operations:

*IF RF CAL ITEM IS LESS THAN X
EXECUTE INSTRUCTION*

*ELSE**EXECUTE INSTRUCTION*

A PMRTI can support basic mathematical operations, such as: addition, subtraction, multiplication, and division. As with
5 the system software code sections, the PMRTI code section may be loaded in response to UI prompts, and the wireless device must be reset after the PMRTI is loaded into code storage section 112. Then the PMRTI section can be executed. If the PMRTI code section is associated with any virtual tables or read-write data, an updated
10 patch manager code section will be transported with the PMRTI for installation in the code storage section 112. Alternately, the PMRTI can be kept and processed from the FSS 110. After the handset 104 has executed all the instructions in the PMRTI section, the PMRTI section can be deleted from the FSS 110. Alternately, the PMRTI is
15 maintained for future operations. For example, the PMRTI may be executed every time the wireless device is energized.

PMRTI is a very powerful runtime instruction engine. The handset can execute any instruction delivered to it through the PMRTI environment. This mechanism may be used to support RF
20 calibrations. More generally, PMRTI can be used to remote debug wireless device software when software problems are recognized by the manufacturer or service provider, typically as the result of user complaints. PMRTI can also record data needed to diagnose software problems. PMRTI can launch newly downloaded system applications
25 for data analysis, debugging, and fixes. PMRTI can provide RW data based updates for analysis and possible short term fix to a problem in

0956743-031001

lieu of an updated system software code section. PMRTI can provide memory compaction algorithms for use by the wireless device.

In some aspects of the invention, the organization of the system software into symbol libraries may impact the size of the volatile memory 114 and nonvolatile memory 108 required for execution. This is due to the fact that the code sections are typically larger than the symbol libraries arranged in the code sections. These larger code sections exist to accommodate updated code sections. Organizing the system software as a collection of libraries impacts the nonvolatile memory size requirement. For the same code size, the amount of nonvolatile memory used will be higher due to the fact that code sections can be sized to be larger than the symbol libraries arranged within.

Once software updates have been delivered to the wireless device, the software maintenance system 100 supports memory compaction. Memory compaction is similar to disk de-fragmentation applications in desktop computers. The compaction mechanism ensures that memory is optimally used and is well balanced for future code section updates, where the size of the updated code sections are unpredictable. The system 100 analyzes the code storage section as it is being patched (updated). The system 100 attempts to fit updated code sections into the memory space occupied by the code section being replaced. If the updated code section is larger than the code section being replaced, the system 100 compacts the code sections in memory 112. Alternately, the compaction can be calculated by the

manufacturer or service provider, and compaction instructions can be transported to the wireless device 104.

Compaction can be a time consuming process owing to the complexity of the algorithm and also the vast volume of data movement. The compaction algorithm predicts feasibility before it begins any processing. UI prompts can be used to apply for permission from the user before the compaction is attempted.

In some aspects of the invention, all the system software code sections can be updated simultaneously. A complete system software upgrade, however, would require a larger FSS 110.

Fig. 3 is a schematic block diagram illustrating the present invention dynamic instruction set execution in a wireless communications device. The system 300 comprises a code storage section 112 in memory 108 including executable wireless device system software differentiated into a plurality of current code sections. Code section one (302), code section two (304), code section n (306), and a patch manager code section 308 are shown. However, the invention is not limited to any particular number of code sections. Further, the system 300 further comprises a first plurality of symbol libraries arranged into the second plurality of code sections. Shown are symbol library one (310) arranged in code section one (302), symbol libraries two (312) and three (314) arranged in code section two (304), and symbol library m (316) arranged in code section n (306). Each library comprises symbols having related functionality. For example, symbol library one (310) may be involved in the operation of the wireless device liquid crystal display (LCD). Then, the

symbols would be associated with display functions. As explained in detail below, additional symbol libraries are arranged in the patch manger code section 308.

Fig. 4 is a schematic block diagram of the wireless device memory. As shown, the memory is the code storage section 112 of Fig. 1. The memory is a writeable, nonvolatile memory, such as Flash memory. It should be understood that the code sections need not necessarily be stored in the same memory as the FSS 110. It should also be understood that the present invention system software structure could be enabled with code sections stored in a plurality of cooperating memories. The code storage section 112 includes a second plurality of contiguously addressed memory blocks, where each memory block stores a corresponding code section from the second plurality of code sections. Thus, code section one (302) is stored in a first memory block 400, code section two (304) in the second memory block 402, code section n (306) in the n th memory block 404, and the patch manager code section (308) in the p th memory block 406.

Contrasting Figs. 3 and 4, the start of each code section is stored at corresponding start addresses in memory, and symbol libraries are arranged to start at the start of code sections. That is, each symbol library begins at a first address and runs through a range of addresses in sequence from the first address. For example, code section one (302) starts at the first start address 408 (marked with "S") in code storage section memory 112. In Fig. 3, symbol library one (310) starts at the start 318 of the first code section.

Likewise code section two (304) starts at a second start address 410 (Fig. 4), and symbol library two starts at the start 320 of code section two (Fig. 3). Code section n (306) starts at a third start address 412 in code storage section memory 112 (Fig. 4), and symbol library m (316) starts at the start of code section n 322 (Fig. 3). The patch manager code section starts at pth start address 414 in code storage section memory 112, and the first symbol library in the patch manager code section 308 starts at the start 324 of the patch manager code section. Thus, symbol library one (310) is ultimately stored in the first memory block 400. If a code section includes a plurality of symbol libraries, such as code section two (304), the plurality of symbol libraries are stored in the corresponding memory block, in this case the second memory block 402.

In Fig. 3, the system 300 further comprises a code section address table 326 as a type of symbol included in a symbol library arranged in the patch manager code section 308. The code section address table cross-references code section identifiers with corresponding code section start addresses in memory.

Fig. 5 is a table representing the code section address table 326 of Fig. 3. The code section address table 326 is consulted to find the code section start address for a symbol library. For example, the system 300 seeks code section one when a symbol in symbol library one is required for execution. To find the start address of code section one, and therefore locate the symbol in symbol library one, the code section address table 326 is consulted. The arrangement of symbol libraries in code sections, and the tracking of code sections

with a table permits the code sections to be moved or expanded. The expansion or movement operations may be needed to install upgraded code sections (with upgraded symbol libraries).

Returning to Fig. 3, it should be noted that not every symbol library necessarily starts at the start of a code section. As shown, symbol library three (314) is arranged in code section two (304), but does not start of the code section start address 320. Thus, if a symbol in symbol library three (314) is required for execution, the system 300 consults the code section address table 326 for the start address of code section two (304). As explained below, a symbol offset address table permits the symbols in symbol library three (314) to be located. It does not matter that the symbols are spread across multiple libraries, as long as they are retained with the same code section.

As noted above, each symbol library includes functionally related symbols. A symbol is a programmer-defined name for locating and using a routine body, variable, or data structure. Thus, a symbol can be an address or a value. Symbols can be internal or external. Internal symbols are not visible beyond the scope of the current code section. More specifically, they are not sought by other symbol libraries, in other code sections. External symbols are used and invoked across code sections and are sought by libraries in different code sections. The symbol offset address table typically includes a list of all external symbols.

25 For example, symbol library one (310) may generate
characters on a wireless device display. Symbols in this library

would, in turn, generate telephone numbers, names, the time, or other display features. Each feature is generated with routines, referred to herein as a symbol. For example, one symbol in symbol library one (310) generates telephone numbers on the display. This symbol is represented by an "X", and is external. When the wireless device receives a phone call and the caller ID service is activated, the system must execute the "X" symbol to generate the number on the display. Therefore, the system must locate the "X" symbol.

Fig. 6 is a detailed depiction of symbol library one (310) of Fig. 3, with symbols. Symbols are arranged to be offset from respective code section start addresses. In many circumstances, the start of the symbol library is the start of a code section, but this is not true if a code section includes more than one symbol library. Symbol library one (310) starts at the start of code section one (see Fig. 3). As shown in Fig. 6, the "X" symbol is located at an offset of (03) from the start of the symbol library and the "Y" symbol is located at an offset of (15). The symbol offset addresses are stored in a symbol offset address table 328 in the patch manager code section (see Fig. 3).

Fig. 7 is a table representing the symbol offset address table 328 of Fig. 3. The symbol offset address table 328 cross-references symbol identifiers with corresponding offset addresses, and with corresponding code section identifiers in memory. Thus, when the system seeks to execute the "X" symbol in symbol library one, the symbol offset address table 328 is consulted to locate the exact address of the symbol, with respect to the code section in which it is arranged.

Returning to Fig. 3, the first plurality of symbol libraries typically all include read-write data that must be consulted or set in the execution of these symbol libraries. For example, a symbol library may include an operation dependent upon a conditional statement.

- 5 The read-write data section is consulted to determine the status required to complete the conditional statement. The present invention groups the read-write data from all the symbol libraries into a shared read-write section. In some aspects of the invention, the read-write data 330 is arranged in the patch manager code section 308.
- 10 Alternately (not shown), the read-write data can be arranged in a different code section, code section n (306), for example.

- The first plurality of symbol libraries also includes symbol accessor code arranged in a code section to calculate the address of a sought symbol. The symbol accessor code can be arranged and
- 15 stored at an address in a separate code section, code section two (304), for example. However, as shown, the symbol accessor code 332 is arranged and stored at an address in the patch manager code section 308. The system 300 further comprises a first location for storage of the symbol accessor code address. The first location can be
- 20 a code section in the code storage section 112, or in a separate memory section of the wireless device (not shown). The first location can also be arranged in the same code section as the read-write data. As shown, the first location 334 is stored in the patch manager code section 308 with the read-write data 330, the symbol offset address
- 25 table 328, the code section address table 326, and the symbol accessor code 332, and the patch library (patch symbol library) 336.

0492731-031001
The symbol accessor code accesses the code section address table and symbol offset address tables to calculate, or find the address of a sought symbol in memory. That is, the symbol accessor code calculates the address of the sought symbol using a
5 corresponding symbol identifier and a corresponding code section identifier. For example, if the "X" symbol in symbol library one is sought, the symbol accessor is invoked to seek the symbol identifier (symbol ID) "X_1", corresponding to the "X" symbol (see Fig. 7). The symbol accessor code consults the symbol offset address table to
10 determine that the "X_1" symbol identifier has an offset of (03) from the start of code section one (see Fig. 6). The symbol accessor code is invoked to seek the code section identifier "CS_1", corresponding to code section one. The symbol accessor code consults the code section address table to determine the start address associated with code
15 section identifier (code section ID) "CS_1". In this manner, the symbol accessor code determines that the symbol identifier "X_1" is offset (03) from the address of (00100), or is located at address (00103).

The symbol "X" is a reserved name since it is a part of the actual code. In other words, it has an absolute data associated with
20 it. The data may be an address or a value. The symbol identifier is an alias created to track the symbol. The symbol offset address table and the code section address table both work with identifiers to avoid confusion with reserved symbol and code section names. It is also possible that the same symbol name is used across many symbol
25 libraries. The use of identifiers prevents confusion between these symbols.

Returning to Fig. 1, the system 300 further comprises a read-write volatile memory 114, typically random access memory (RAM). The read-write data 330, code section address table 326, the symbol offset address table 328, the symbol accessor code 332, and the symbol accessor code address 334 are loaded into the read-write volatile memory 114 from the patch manager code section for access during execution of the system software. As is well known, the access times for code stored in RAM is significantly less than the access to a nonvolatile memory such as Flash.

Returning to Fig. 3, it can be noted that the symbol libraries need not necessarily fill the code sections into which they are arranged, although the memory blocks are sized to exactly accommodate the corresponding code sections stored within.

Alternately stated, each of the second plurality of code sections has a size in bytes that accommodates the arranged symbol libraries, and each of the contiguously addressed memory blocks have a size in bytes that accommodates corresponding code sections. For example, code section one (302) may be a 100 byte section to accommodate a symbol library having a length of 100 bytes. The first memory block

would be 100 bytes to match the byte size of code section one.

However, the symbol library loaded into code section 1 may be smaller than 100 bytes. As shown in Fig. 3, code section one (302) has an unused section 340, as symbol library one (310) is less than 100 bytes. Thus, each of the second plurality of code sections may have a size larger than the size needed to accommodate the arranged symbol

libraries. By "oversizing" the code sections, larger updated symbol libraries can be accommodated.

Contiguously addressed memory blocks refers to partitioning the physical memory space into logical blocks of variable size. Code sections and memory blocks are terms that are essentially interchangeable when the code section is stored in memory. The concept of a code section is used to identify a section of code that is perhaps larger than the symbol library, or the collection of symbol libraries in the code section as it is moved and manipulated.

As seen in Fig. 3, the system 300 includes a patch symbol library, which will be referred to herein as patch library 336, to arrange new code sections in the code storage section with the current code sections. The arrangement of new code sections with current code sections in the code storage section forms updated executable system software. The patch manager 336 not only arranges new code sections in with the current code sections, it also replaces code sections with updated code sections.

Returning to Fig. 4, the file system section 110 of memory 108 receives new code sections, such as new code section 450 and updated patch manager code section 452. The file system section also receives a first patch manager run time instruction (PMRTI) 454 including instructions for arranging the new code sections with the current code sections. As seen in Fig. 1, an airlink interface 150 receives new, or updated code sections, as well as the first PMRTI. Although the airlink interface 150 is being represented by an antenna, it should be understood that the airlink interface would also include

an RF transceiver, baseband circuitry, and demodulation circuitry (not shown). The file system section 110 stores the new code sections received via the airlink interface 150. The patch library 336, executing from read-write volatile memory 114, replaces a first code section in the code storage section, code section n (306) for example, with the new, or updated code section 450, in response to the first PMRTI 454. Typically, the patch manager code section 308 is replaced with the updated patch manager code section 452. When code sections are being replaced, the patch library 336 over-writes the first code section, code section n (306) for example, in the code storage section 112 with the updated code sections, code section 450 for example, in the file system section 110. In the extreme case, all the code sections in code storage section 112 are replaced with updated code sections. That is, the FSS 110 receives a second plurality of updated code sections (not shown), and the patch library 336 replaces the second plurality of code sections in the code storage section 112 with the second plurality of updated code sections. Of course, the FSS 110 must be large enough to accommodate the second plurality of updated code sections received via the airlink interface.

As noted above, the updated code sections being received may include read-write data code sections, code section address table code sections, symbol libraries, symbol offset address table code sections, symbol accessor code sections, or a code section with a new patch library. All these code sections, with their associated symbol libraries and symbols, may be stored as distinct and independent

10

15

25



5

10

20

25

time engine for processing the dynamic instruction sets, enabled as run-time library 370. As with the compactor library 342 and patch library 336 mentioned above, the run-time library 370 is typically located in the patch manager code section 308. However, the run-
5 time library 370 could alternately be located in another code section, for example the first code section 304.

The dynamic instruction sets are a single, or multiple sets of instructions that include conditional operation code, and generally include data items. The run-time engine reads the operation code
10 and determines what operations need to be performed. Operation code can be conditional, mathematical, procedural, or logical. The run-time engine, or run-time library 370 processes the dynamic instruction sets to perform operations such as mathematical or logical operations. That is, the run-time engine reads the dynamic
15 instruction set 470 and performs a sequence of operations in response to the operation code. Although the dynamic instruction sets are not limited to any particular language, the operation code is typically a form of machine code, as the wireless device memory is limited and execution speed is important. The operation code is
20 considered conditional in that it analyzes a data item and makes a decision as a result of the analysis. The run-time engine may also determine that an operation be performed on data before it is analyzed.

For example, the operation code may specify that a data
25 item from a wireless device memory be compared to a predetermined value. If the data item is less than the predetermined value, the data

09527131-031001
FOOTED TET 250

item is left alone, and if the data item is greater than the predetermined value, it is replaced with the predetermined value. Alternately, the operation code may add a second predetermined value to a data item from the wireless device memory, before the above-mentioned comparison operation is performed.

As mentioned above, the file system section nonvolatile memory 110 receives the dynamic instruction sets through an interface such as the airlink 150. As shown in Fig. 1, the interface can also be radio frequency (RF) hardline 160. Then, the PMRTI can be received by the FSS 110 without the system software being operational, such as in a factory calibration environment. The PMRTI can also be received via a logic port interface 162 or an installable memory module 164. The memory module 164 can be installed in the wireless device 104 at initial calibration, installed in the field, or installed during factory recalibration. Although not specially shown, the PMRTI can be received via an infrared or Bluetooth interfaces.

Fig. 8 is a depiction of instructions being accessed by the run-time engine 370. Shown is a first instruction 800, a second instruction 802, and a j th instruction 804, however, the dynamic instruction set is not limited to any particular number of instructions. The length of the operation code in each instruction is fixed. The run-time engine 370 captures the length of the instruction, as a measure of bytes or bits, determine if the instruction includes data items. The remaining length of the instruction, after the operation code is subtracted, includes the data items. The run-time engine extracts the data items from the instruction. As shown, the length 806 of the first

05927131-031001
T00T90-TET2690

5 instruction 800.

15 identifier "X_1" is shown in the first instruction 800. The symbol offset address table 328 locates the corresponding symbol in a code section with the "CS_1" identifier and an offset of "3". The code section address table 326 gives the start address of code section one (302). In this manner, the symbol "X" is found (see Fig. 6).

25 Alternately, the “X” symbol can be operation code, and the run-time engine executes the symbol “X” when it is located.

PMRTI can be used to update system data, or system data items. In some aspects of the invention system data is stored in a code section in the file system section 110, code section 472 for example, see Fig. 4. The run-time engine accesses system data from code section 472 and analyzes the system data. The run-time engine processes the operation code of the dynamic instruction sets to perform mathematical or logical operation on data items, as described above. After the operation, the run-time engine processes the instructions to create updated system data. Note that the updated system data may include unchanged data items in some circumstances. The system data in the second code section 472 is replaced with the updated system data in response to the operation code. Thus, by the processing of instruction by the run-time engine, the system software is controlled to execute using the updated system data in code section 472. In this manner, specifically targeted symbols in the system software can be updated, without replacing entire code sections. By the same process, the system data can be replaced in a code section in the code storage section 112. For example, the system data can be stored in the third code section 344, and the run-time engine can replace the system data in the third code section with updated system data in response to the operation code.

PMRTI can also be used to update data items in volatile memory 114. As an example, the volatile memory 114 accept read-write data 330, see Fig. 1. The read-write data can be from one, or from a plurality of code sections in the code storage section 112 and/or the FSS 110. The run-time engine accesses the read-write

data, analyzes the read-write data 330, creates updated read-write data, and replaces the read-write data 330 in the volatile memory 114 with the updated read-write data in response to the operation code. Then, the system software is controlled to execute using the updated
5 read-write data in volatile memory 114.

In some aspects of the invention, the run-time engine monitors the execution of the system software. Performance monitoring is broadly defined to include a great number of wireless device activities. For example, data such as channel parameters,
10 channel characteristics, system stack, error conditions, or a record of data items in RAM through a sequence of operations leading to a specific failure condition or reduced performance condition can be collected. It is also possible to use dynamic instructions sets to analyze collected performance data, provide updated data variants,
15 and recapture data to study possible solutions to the problem. Temporary fixes can also be provisioned using PMRTI processes.

More specifically, the run-time engine collects performance data, and stores the performance data in the file system section in response to the operation code. Then, the system software
20 is controlled to execute by collecting the performance data for evaluation of the system software. Evaluation can occur as a form of analysis performed by dynamic instruction set operation code, or it can be performed outside the wireless device. In some aspects of the invention, the run-time engine accesses the performance data that
25 has been collected from the file system section and transmits the performance data via an airlink interface in response to the operation

THE NEW YORK PUBLIC LIBRARY

In some aspects of the invention, file system section 110 receives a patch manager run time instruction including a new code section. For example, a new code section 474 is shown in Fig. 4. Alternately, the new code section can be independent of the PMRTI, such as new code section *n* (450). For example, the new code section *n* (450) may have been received in earlier airlink communications, or have been installed during factory calibration. The run-time engine adds the new code section 474 (450) to the code storage section in response to the operation code. In some aspects of the invention, the new code section is added to an unused block in the code storage section 112. Alternately, a compaction operation is required. Then, the system software is controlled to execute using the new code section 474 (450). In other aspects of the invention, the PMRTI 454 includes an updated code section 474. Alternately, the new code section 450 is an updated code section independent of the PMRTI. The run-time engine replaces a code section in the code storage section, code section two (304) for an example, with the updated code section 474 (450) in response to the operation code. The system software is controlled to execute using the updated code section 474 (450). In some aspects of the invention a compaction operation is required to accommodate the updated code section. Alternately, the updated code section is added to an unused or vacant section of the code storage section.

As explained above, the addition of a new code section or the updating of a code section typically requires the generation of a new code section address table, as these operation involve either new and/or changed code section start addresses. Further, a compaction operation also requires a new code section address table. The compaction operations may be a result of the operation of the compactor 342, explained above, or the result of PMRTI instructions that supply details as to how the compaction is to occur. When the PMRTI includes downloading and compaction instructions, the PMRTI typically also includes a new code section address table that becomes valid after the downloading and compaction operations have been completed.

Figs. 10a and 10b are flowcharts illustrating the present invention method for executing dynamic instruction sets in a wireless communications device. Although depicted as a sequence of numbered steps for clarity, no order should be inferred from the numbering (and the numbering in the methods presented below) unless explicitly stated. The method starts at Step 1000. Step 1001a forms the system software into symbol libraries, each symbol library comprising symbols having related functionality. Step 1001b arranges the symbol libraries into code sections. Step 1002 executes system software. Step 1003 receives the dynamic instruction sets. Receiving the dynamic instruction sets in Step 1003 includes receiving the dynamic instruction sets through an interface selected from the group including airlink, radio frequency (RF) hardline, installable memory module, infrared, and logic port interfaces. In

	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415	2416	2417	2418	2419	2420	2421	2422	2
--	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	---

5 launching a run-time engine includes invoking a run-time library
from a first code section. The run-time engine can be launched from
either volatile or nonvolatile memory. Step 1006 processes dynamic
instruction sets. Processing dynamic instruction sets includes
processing instructions in response to mathematical and logical
10 operations. In some aspects of the invention, Step 1007 (not shown),
following the processing of the dynamic instruction sets, deletes
dynamic instruction sets. Step 1008 operates on system data and
system software. Step 1010, in response to operating on the system
data and system software, controls the execution of the system
15 software.

20 to read the patch manager run time instruction operation code. Step
1006b performs a sequence of operations in response to the operation
code.

25 of code sections and arranging symbols to be offset from their
respective code section start addresses. Then the method comprises

5

10

20

25

Processing dynamic instruction sets in Step 1106 includes substeps. Step 1106a accesses system data stored in a second code section in the file system section. Step 1106b analyzes the system data. Step 1106c creates updated system data. Then, operating on system data and system software in Step 1108 includes replacing the system data in the second section with the updated system data, and controlling the execution of the system software in Step 1010 includes using the updated system data in the execution of the system software.

Fig. 12 is a flowchart illustrating another exemplary dynamic instruction set operation. Several of the Steps in Fig. 12 are the same as in Fig. 10, and are not repeated here in the interest of brevity. Step 1201c stores a plurality of code sections in a code storage section nonvolatile memory. Processing dynamic instruction sets in Step 1206 includes substeps. Step 1206a accesses system data stored in a third code section in the code storage section (CSS). Step 1206b analyzes the system data. Step 1206c creates updated system data. Operating on the system data and system software in Step 1208 includes replacing the system data in the third code section with the updated system data. Controlling the execution of the system software in Step 1210 includes using the updated system data in the execution of the system software.

Fig. 13 is a flowchart illustrating a third exemplary dynamic instruction set operation. Several of the Steps in Fig. 13 are the same as in Fig. 10, and are not repeated here in the interest of brevity. Step 1301c stores a plurality of code sections in a code storage section nonvolatile memory. Step 1301d loads read-write data

into volatile memory. Processing dynamic instruction sets in Step 1306 includes substeps. Step 1306a accesses the read-write data in volatile memory. Step 1306b analyzes the read-write data. Step 1306c creates updated read-write data. Operating on the system data and system software in Step 1308 includes replacing the read-write data in volatile memory with the updated read-write data. Controlling the execution of the system software includes using the updated read-write data in the execution of the system software.

Fig. 14 is a flowchart illustrating a fourth exemplary dynamic instruction set operation. Several of the Steps in Fig. 14 are the same as in Fig. 10, and are not repeated here in the interest of brevity. Processing dynamic instruction sets includes substeps. Step 1406a, in response to the operation code, monitors the execution of the system software. Step 1406b collects performance data. Step 1406c stores the performance data. Step 1406d transmits the stored data via an airlink interface. Operating on the system data and system software in Step 1408 includes using the performance data in the evaluation of system software.

Fig. 15 is a flowchart illustrating a fifth exemplary dynamic instruction set operation. Several of the Steps in Fig. 15 are the same as in Fig. 10, and are not repeated here in the interest of brevity. Step 1501c stores a plurality of code sections in a code storage section nonvolatile memory. Receiving patch manager run time instructions in Step 1503 includes receiving a new code section. Operating on the system data and system software in Step 1508 includes adding the new code section to the code storage section, and

controlling the execution of the system software in Step 1510 includes using the new code section in the execution of the system software.

Alternately, receiving a new code section in Step 1503 includes receiving an updated code section. Then, operating on the
5 system data and system software in Step 1508 includes replacing a fourth code section in the code storage section with the updated code section.

A system and method have been provided for executing dynamic instruction sets in a wireless communications device, so as
10 to aid in the process of updating the software and monitoring the performance of the software. The system is easily updateable because of the arrangement of symbol libraries in code sections, with tables to access the start addresses of the code sections in memory and the offset addresses of symbols in the symbol libraries. The use on
15 dynamic instruction sets permits custom modifications to be performed to each wireless device, based upon specific characteristics of that device. A few general examples have been given illustrating possible uses for the dynamic instructions sets. However, the present invention is not limited to just these examples. Other variations and
20 embodiments of the invention will occur to those skilled in the art.

WE CLAIM: